Algoritmos y Estructura de Datos: Examen Julio (Solución)

Grados Ing. Inf. y Mat. Inf. Julio 2014

Departamento de Lenguajes, Sistemas Informáticos e Ingeniería de Software

Apellidos: .....

Nombre: .....

DNI / NIE: ..... **Núm. matrícula:** .....

Normas

- Deben rellenarse los campos obligatorios **apellidos, nombre, y DNI/NIE**.
- Este examen consta de **5 preguntas** en **6 páginas**.
- La puntuación total del examen es de **10 puntos**.
- La duración total del examen es de **90 minutos**.
- El examen debe contestarse **en las hojas que se proporcionan**.
- Las calificaciones provisionales de este examen se publicarán en el Aula Virtual el **XXX** junto con las soluciones. La fecha de la revisión de este examen es el **XXX**. La hora y lugar de dicha revisión se anunciará en el Aula Virtual.

(2 puntos) 1. **Se pide:** Implementar en Java el método

```
public static <E> PositionList<E> ancestros(Tree<E> t, Position<E> p)
```

que toma como parámetro un árbol no vacío `t` y un nodo `p` de dicho árbol. El método debe devolver como resultado una lista de posiciones con los elementos en los ancestros propios de `p`. El interfaz `Tree<E>` está disponible en el Apéndice A.1.

```
public static <E> PositionList<E> ancestros(Tree<E> t, Position<E> p) {
  PositionList<E> r = new NodePositionList<E>();
  while (!t.isRoot(p)) {
    p = t.parent(p);
    r.addLast(p.element());
  }
  return r;
}
```

(1 punto) 2. **Se pide:** Indicar la complejidad en caso peor del método `ancestros` de la pregunta anterior.

En el caso peor el nodo `p` será el nodo hoja de mayor profundidad. La profundidad del árbol se define como la de su nodo hoja de mayor profundidad, y dicha cantidad coincide con la altura h del árbol. Por tanto la complejidad en caso peor es $O(h)$.

(1 punto) 3. Se tiene la cabecera del método `lineal` que toma como argumento un entero:

```
public void lineal(int n)
```

Se pide: Inventar código para dicho método que tenga en el caso peor complejidad lineal en el tamaño del entero.

```
public void lineal(int n) {
  n = Math.abs(n);
  for (int i = 0; i < n; i++)
    ;
}
```

(4 puntos) 4. Se desea implementar una clase de objetos comparadores que comparen objetos iteradores:

```
public class IteratorComparator<E> implements Comparator<Iterator<E>> {
  private Comparator<E> compElem;

  public IteratorComparator(Comparator<E> compElem) {
    this.compElem = compElem;
  }

  public int compare(Iterator<E> it1, Iterator<E> it2) {
    // COMPLETAR
  }
}
```

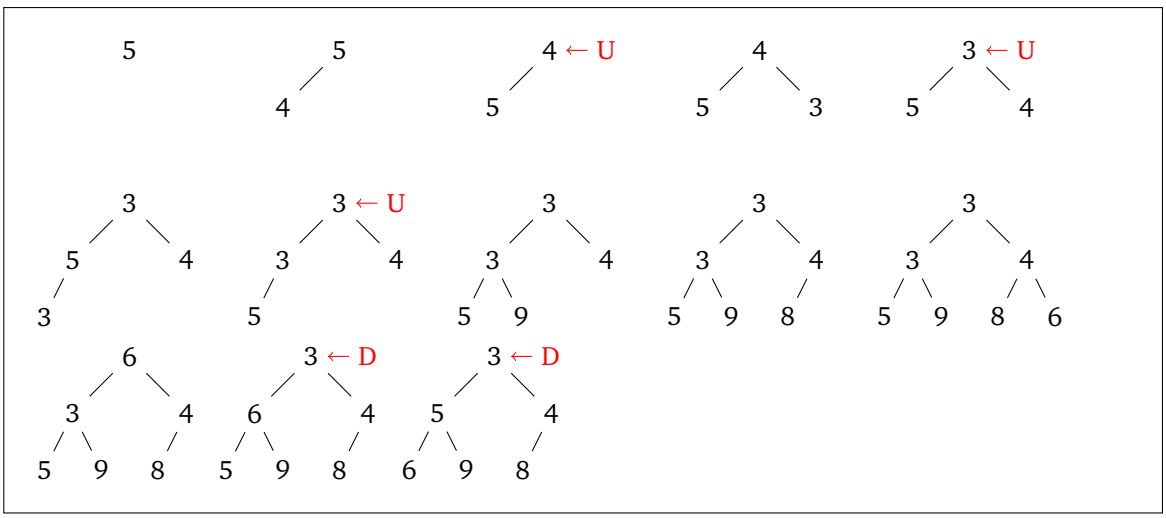
El constructor de la clase toma como parámetro un comparador de elementos `compElem`. **Se pide:** Implementar el método `compare` que debe devolver el resultado de comparar los iteradores `it1` e `it2` que toma como parámetros. La comparación de dichos iteradores se realiza comparando los elementos que devuelven `it1.next()` e `it2.next()` con el comparador de elementos. En el momento en el que la comparación de elementos no sea 0, el resultado de la comparación de iteradores será el de dicha comparacion de elementos. Si todas las comparaciones de elementos devuelven 0, se considera «mayor» el iterador al que le quedan elementos por recorrer.

```
public int compare(Iterator<E> it1, Iterator<E> it2) {
  int i = 0;
  while (it1.hasNext() && it2.hasNext() &&
         (i = compElem.compare(it1.next(),it2.next())) == 0)
    ;
  if (i==0 && (it1.hasNext() || it2.hasNext()))
    i = it1.hasNext() ? 1 : -1 ;
  return i;
}
```

(2 puntos) 5. Se tiene una cola con prioridad vacía implementada mediante un montículo sobre la que se llevan a cabo las siguientes operaciones:

- Se insertan de forma consecutiva las siguientes entradas (sólo mostramos las claves de tipo `Integer`, obviaremos los elementos): 5, 4, 3, 3, 9, 8, 6.
- Inmediatamente después se borra la entrada con clave mínima.

La cola con prioridad utiliza internamente el comparador estándar de enteros. **Se pide:** Dibujar los árboles (casi)completos que conforman el montículo en cada paso (añadir la entrada al árbol, «up-heap», «down-heap», borrar la entrada del árbol), etiquetando los árboles de los pasos de «up-heap» con la letra **U** y los árboles de los pasos de «down-heap» con la letra **D**.



A. Código de apoyo

A.1. Interfaz `Tree<E>`

```
package net.datastructures;
import java.util.Iterator;
/**
 * An interface for a tree where nodes can have an arbitrary number of children.
 * @author Michael Goodrich
 */
public interface Tree<E> {
  /** Returns the number of nodes in the tree. */
  public int size();

  /** Returns whether the tree is empty. */
  public boolean isEmpty();

  /** Returns an iterator of the elements stored in the tree. */
  public Iterator<E> iterator();

  /** Returns an iterable collection of the the nodes. */
  public Iterable<Position<E>> positions();

  /** Replaces the element stored at a given node. */
  public E replace(Position<E> v, E e) throws InvalidPositionException;

  /** Returns the root of the tree. */
  public Position<E> root() throws EmptyTreeException;

  /** Returns the parent of a given node. */
  public Position<E> parent(Position<E> v)
    throws InvalidPositionException, BoundaryViolationException;

  /** Returns an iterable collection of the children of a given node. */
  public Iterable<Position<E>> children(Position<E> v)
    throws InvalidPositionException;

  /** Returns whether a given node is internal. */
  public boolean isInternal(Position<E> v) throws InvalidPositionException;

  /** Returns whether a given node is external. */
  public boolean isExternal(Position<E> v) throws InvalidPositionException;

  /** Returns whether a given node is the root of the tree. */
  public boolean isRoot(Position<E> v) throws InvalidPositionException;
}
```

A.2. Interfaz `PositionList<E>`

```
package net.datastructures;
import java.util.Iterator;
/**
 * An interface for positional lists.
 * @author Roberto Tamassia, Michael Goodrich
 */

public interface PositionList<E> extends Iterable<E> {
  /** Returns the number of elements in this list. */
  public int size();

  /** Returns whether the list is empty. */
  public boolean isEmpty();

  /** Returns the first node in the list. */
  public Position<E> first();

  /** Returns the last node in the list. */
  public Position<E> last();

  /** Returns the node after a given node in the list. */
  public Position<E> next(Position<E> p)
    throws InvalidPositionException, BoundaryViolationException;

  /** Returns the node before a given node in the list. */
  public Position<E> prev(Position<E> p)
    throws InvalidPositionException, BoundaryViolationException;

  /** Inserts an element at the front of the list, returning new position. */
  public void addFirst(E e);

  /** Inserts and element at the back of the list, returning new position. */
  public void addLast(E e);

  /** Inserts an element after the given node in the list. */
  public void addAfter(Position<E> p, E e) throws InvalidPositionException;

  /** Inserts an element before the given node in the list. */
  public void addBefore(Position<E> p, E e) throws InvalidPositionException;

  /** Removes a node from the list, returning the element stored there. */
  public E remove(Position<E> p) throws InvalidPositionException;

  /** Replaces the element stored at the given node, returning old element. */
  public E set(Position<E> p, E e) throws InvalidPositionException;

  /** Returns an iterable collection of all the nodes in the list. */
  public Iterable<Position<E>> positions();

  /** Returns an iterator of all the elements in the list. */
  public Iterator<E> iterator();
}
```